# TP – Les marmottes de Huffman

## I - Les marmottes au sommeil léger

Cette partie du TP est tirée du travail de Marie Duflot-Kremer « Les marmottes au sommeil léger ».

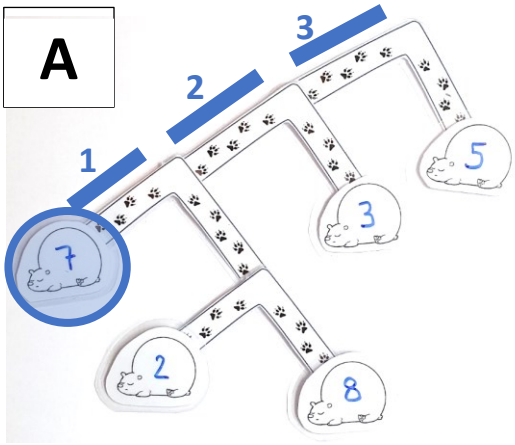
### 1) Réalisation du terrier

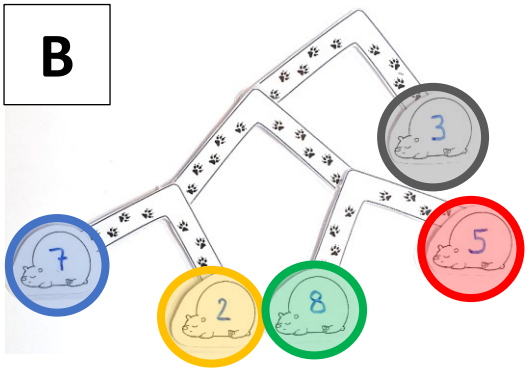
Les marmottes doivent se préparer un nouveau terrier pour passer l'hiver. A vous de les aider !
Ces animaux sont très doués pour creuser, mais leur terrier doit répondre à certaines contraintes :

- ✓ A partir de l'entrée du terrier (située en haut), les marmottes creusent deux tunnels perpendiculaires vers le bas, puis au bout de ces tunnels, soit elles installent une chambre pour l'une d'elle, soit un autre jeu de deux couloirs et ainsi de suite.
- ✓ Chaque marmotte se lève un certain nombre de fois pendant l'hiver (nombre noté en bleu) pour sortir du terrier, faire ses besoins et retourner se coucher. Pour ne pas réveiller ses congénères qui ont le sommeil léger, la marmotte doit parcourir le chemin le plus court possible jusqu'à la sortie.

Vous devez proposer l'organisation de terrier (galeries et positions des chambres de chaque marmotte) qui minimise les déplacements, c'est-à-dire dont le nombre total de sections de couloir parcourues par les marmottes au cours de l'hiver soit le plus faible possible.
Pour cela, on va calculer le « score » d'un terrier en faisant la somme du produit :
<nombres de réveils de la marmotte> × <nombres de sections de couloir à traverser de sa chambre jusqu'à la sortie>
pour chaque marmotte.

Pour le terrier A ci-contre, la marmotte en bas à gauche se réveille 7 fois et parcours 3 sections de couloir avant d'arriver à la sortie, soit $7 \times 3 = 21$ déplacements pendant l'hiver.
Au total le score de ce terrier est donc de :
$(7 \times 3) + (2 \times 4) + (8 \times 4) + (3 \times 2) + (5 \times 1) = 72$




Pour le terrier B, le score est alors de :
$(7 \times 3) + (2 \times 3) + (8 \times 3) + (3 \times 1) + (5 \times 3) = 69$

**Question 1 :**
A l'aide du matériel fourni, essayer de trouver le terrier ayant le plus petit score.
Représentez alors votre terrier sur votre feuille et notez son score. Quelle a été votre stratégie ?

### 2) Un algorithme

On aimerait maintenant systématiser le processus car les configurations de marmottes peuvent varier et on voudrait être certain d'avoir le meilleur terrier possible.

**Question 2 :**
Essayer de trouver un algorithme permettant de construire le terrier optimal. Décrire cet algorithme en langage naturel ou en pseudo-code.

Vous devez faire valider votre algorithme ou attendre l'exposé de l'algorithme par le professeur avant de continuer.

## II - Le codage de Huffman

Il a été prouvé que l'algorithme vu avec les marmottes est l'algorithme optimal pour le codage de symboles uniques, c'est-à-dire qu'on ne peut pas faire plus court que ce codage. Cet algorithme a été inventé par l'informaticien américain David Albert Huffman, et publié en 1952.
Étant optimal il est utilisé dans de très nombreuses application de compression de données, comme le célèbre format zip (ou rar). En fait il constitue souvent la dernière étape de la compression (on effectue d'abord une première compression qui dépend de la nature des données (son, image, texte, vidéo…) puis on stocke les données compressées avec l'algorithme d'Huffman).
Le but de cette deuxième partie est d'écrire un programme capable d'effectuer la compression et la décompression d'un texte avec l'algorithme de Huffman et d'estimer son efficacité.

La compression d'un texte avec l'algorithme de Huffman se fait en quatre étapes :

① Analyse du texte et détermination des fréquences des différents symboles constituant le texte

② Création de l'arbre binaire de codage en suivant l'algorithme vu avec les marmottes

③ Création d'un dictionnaire de codage associant à chaque symbole son code binaire dans l'arbre

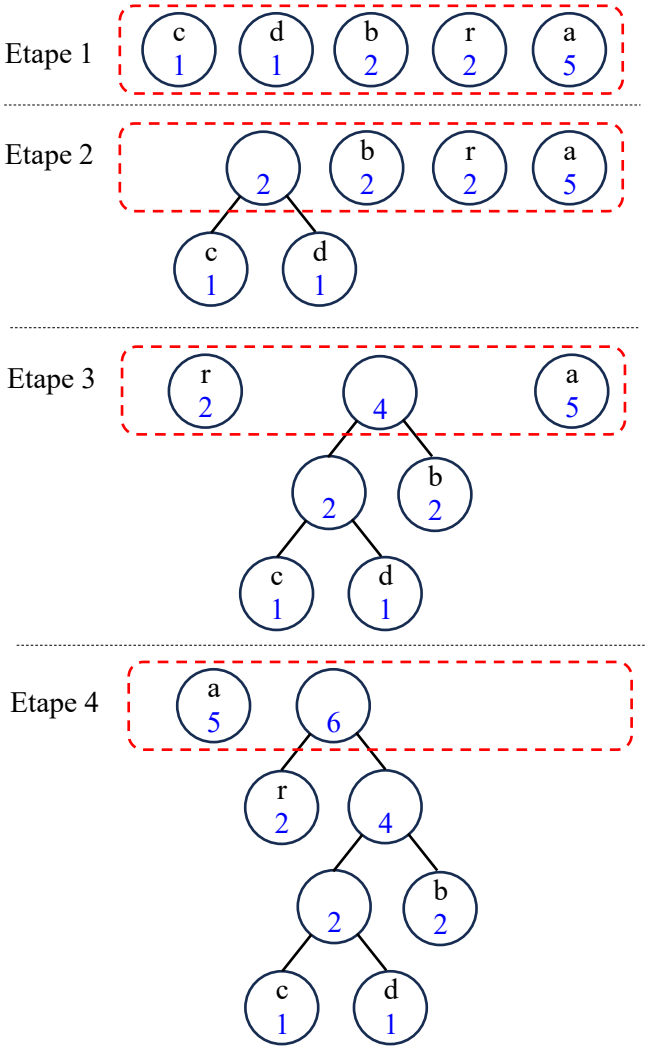
④ Encodage du texte, caractère par caractère en utilisant le dictionnaire de codage

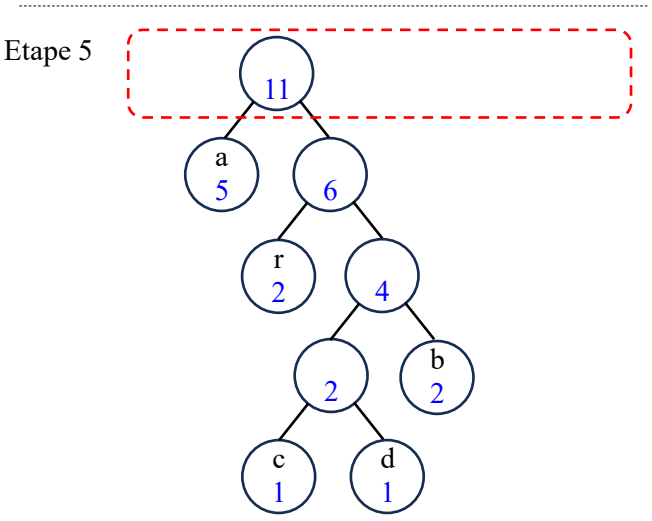
Exemple avec le texte très simple « abracadabra » :

① Fréquence des symboles : 'c': 1, 'd': 1, 'b': 2, 'r': 2, 'a': 5

② Etapes de la création de l'arbre :

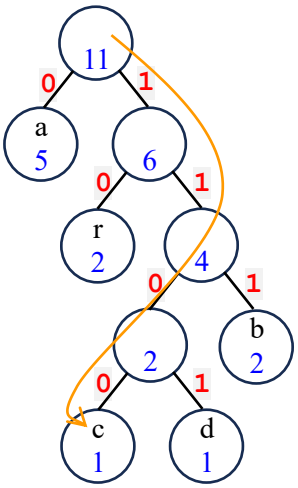
A chaque étape les arbres dans la liste des symboles à traiter (encadrés en pointillés rouges) sont reclassés par fréquences croissantes jusqu'à ce qu'il n'en reste plus qu'un qui est alors la racine de l'arbre cherché.



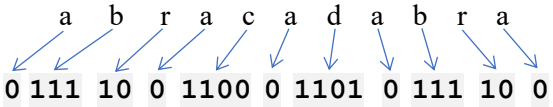



# TP – Les marmottes de Huffman

③ On explore l'arbre jusqu'à chaque feuille pour retrouver le code de chaque symbole : à chaque fois qu'on passe par le fils gauche on ajoute un « `0` » au code et si on passe par le fils droit, c'est un « `1` » qui est rajouté. Ainsi dans l'exemple ci-contre, le code associé au symbole « c » est `1100`.
On obtient le dictionnaire :
`{'a': '0', 'r': '10', 'c': '1100', 'd': '1101', 'b': '111'}`




④ Encodage du texte :




Le texte encodé est donc : `01111001100011010111100`

## 1) Programmation de la classe `Arbre`

On souhaite programmer une classe `Arbre` qui s'appuie sur la classe `Noeud` ci-dessous et permet de gérer des arbres binaires avec deux étiquettes (une pour le symbole et une pour la fréquence).

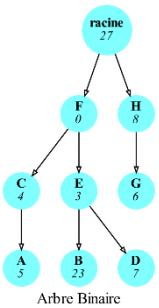
```
class Noeud:
    def __init__(self,symbole,frequence,gauche=None,droit=None):
        self.symbole = symbole
        self.frequence = frequence
        self.brancheg=gauche
        self.branched=droit

    def __lt__(self, autre) -> bool:
        if type(autre) != Noeud: raise TypeError
        if self.frequence == autre.frequence: return self.symbole < autre.symbole
        return self.frequence < autre.frequence
```

**Question 3 :**
A quoi sert la méthode `__lt__` de la classe `Noeud` ? Indiquer ce qu'elle fait et proposer un ou deux exemples d'application.

**Question 4 :**
Dans un fichier « arbre.py » programmer la classe `Arbre` dont l'interface est donnée ci-contre. On pourra utiliser le fichier arbre.py fourni et le compléter.
On rajoutera ensuite une méthode `__lt__` à la classe de manière à pouvoir trier un tableau d'arbres.

| Méthode | Description |
|---|---|
| *constructeur* | Crée un objet Arbre éventuellement vide |
| est_vide() | Renvoie `True` si l'arbre est vide |
| get_symbole() | Renvoie le symbole de la racine de l'arbre |
| get_frequence() | Renvoie la fréquence de la racine de l'arbre |
| get_gauche() | Renvoie le nœud correspondant au sous-arbre gauche |
| get_droite() | Renvoie le nœud correspondant au sous-arbre droit |
| get_racine() | Renvoie le nœud racine de l'arbre |
| est_feuille() | Renvoie `True` si la racine de l'arbre est une feuille |

On pourra ensuite visualiser graphiquement les arbres crées avec la commande `VizuArbreB(arbre)` si on a importé le fichier « vizu_arbreb.py ».
Ainsi l'arbre proposé comme exemple de test dans le fichier « arbre.py » donnera la représentation ci-contre.



Arbre Binaire

# TP – Les marmottes de Huffman

## 2) Algorithme de Huffman

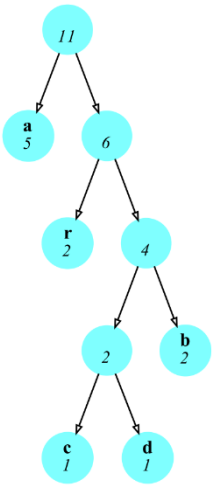
On va maintenant s'occuper de programmer l'algorithme de Huffman en écrivant des fonctions pour chacune des étapes.

On écrira ces fonctions dans un autre fichier « Huffman.py » par exemple qui importera les classes du fichier « arbre.py ».

**Question 5 :**
Ecrire une fonction `cree_dictionnaire_de_frequence(texte:str) -> dict` qui prend en argument une chaîne représentant le texte à coder et renvoie un dictionnaire dont les clés sont les symboles présents dans le texte et les valeurs les fréquences d'apparitions de ces symboles dans le texte.

`cree_dictionnaire_de_frequence("abracadabra")` doit renvoyer :
`{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}`

**Question 6 :**
Ecrire une fonction `cree_arbre_huffman(dico:dict) -> Arbre` qui renvoie l'arbre de Huffman correspondant au dictionnaire de fréquence fournit en argument.
`cree_arbre_huffman({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})` doit renvoyer l'arbre représenté ci-contre.




Rappel : On peut trier un tableau `tab` par ordre croissant avec `tab.sort()`.

**Question 7 :**
Ecrire une fonction `cree_dictionnaire_symbole_code(arbre:Arbre) -> dict:` qui renvoie un dictionnaire associant chaque symbole son code binaire. Le code binaire sera ici donné sous la forme d'une chaîne de caractère contenant des `'0'` et des `'1'`.

`cree_dictionnaire_symbole_code(arbre)` avec l'arbre précédent (celui de `"abracadabra"`) doit renvoyer : `{'a': '0', 'r': '10', 'c': '1100', 'd': '1101', 'b': '111'}`

**Question 8 :**
Ecrire une fonction `codage_huffman(texte:str, dico:dict) -> str` qui renvoie une chaine de 0 et de 1 correspondant au codage de Huffman du texte donné en argument et dont on donne le dictionnaire symbole-code.

`codage_huffman("abracadabra",{'a': '0', 'r': '10', 'c': '1100', 'd': '1101', 'b': '111'})` doit donner `"01111001100011010111100"`.

# TP – Les marmottes de Huffman

**Question 9 :**
Regrouper les fonctions précédentes pour écrire une fonction `compression_huffman(texte:str) -> str` qui renvoie une chaine de 0 et de 1 correspondant au codage de Huffman du texte donné en argument.
Ainsi `codage_huffman("abracadabra")` doit donner `"011110011000110101111100"`.

## 3) Efficacité de la compression

Maintenant qu'on dispose d'une fonction pour effectuer la compression de Huffman d'un texte quelconque, on va essayer de déterminer son efficacité sur un texte réel.
Pour cela, on va utiliser le fichier « swann.txt » qui contient le roman « Du côté de chez Swann » de Marcel Proust, issu du site http://gutenberg.org/.

**Question 10 :**
Ecrire les bouts de code permettant de lire le fichier « swann.txt » et d'en faire la compression avec la méthode de Huffman. Calculer alors le taux de compression sur ce texte et conclure.

Taux de compression = $\frac{taille\_original - taille\_compressée}{taille\_original} \times 100$

La taille de l'original s'exprime en bits et correspond au nombre de caractères multiplié par 8 (nombre de bits par symbole si on a un codage de type ANSI). La taille compressée correspond au nombre de bits obtenus avec la compression de Huffman.

Vous pouvez essayer avec d'autres œuvres téléchargées sur le site du projet Gutenberg et comparer à ce résultat.

## 4) Décompression

On cherche à valider notre algorithme de compression en décompressant le texte compressé et vérifier qu'on retrouve bien le texte original (l'algorithme de Huffman correspond bien sûr à une compression sans perte).

**Question 11 :**
Ecrire une fonction `decompression_huffman(texte_code:str,racine:Arbre) -> str` qui prend en argument la chaîne codée par la méthode de Huffman et l'arbre de codage puis renvoie le texte décodé (texte original).
Vérifier que votre fonction permet de retrouver le texte original. Notamment que `decompression_huffman("011110011000110101111100", arbre)` avec l'arbre utilisé pour l'exemple `"abracadabra"` permet bien de retrouver la chaîne `"abracadabra"`.

## 5) Les limites du TP

Dans un souci de simplification, nous avons ici préféré travailler avec des chaînes de `'0'` et de `'1'` plutôt qu'avec des bits, ce qui fait que si on souhaitait enregistrer le fichier compressé il prendrait plus de place que l'original, mais dans les cas réels, on peut utiliser les `bytearray` pour créer un vrai flux binaire et enregistrer le fichier compressé en mode binaire.

**Question 12 :**
D'après vous quelles sont les limites de la méthode de codage vue ici ? Est-il adapté à toutes les données ? A toutes les situations ?